Department of Computer Science and Engineering
Indian Institute of Technology Guwahati

CS348

Assignment - 7: Parser for microC

1 Preamble – micro C

This assignment follows the lexical specification of C language from the International Standard **ISO/IEC 9899:1999 (E)** with some minor modifications. To keep the assignment simple, we have chosen a subset of the specification as given below. We shall refer to this language as micro C.

The lexical specification quoted here is written using a precise yet compact notation typically used for writing language specifications. We first outline the notation and then present the Lexical Grammar that we shall work with.

2 Notation

In the syntax notation used here, syntactic categories (non-terminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (*:*) following a non-terminal introduces its definition. Alternative definitions are listed on separate lines, except when prefaced by the words "one of". An optional symbol is indicated by the subscript "opt", so that the following indicates an optional expression enclosed in braces.

begin $expression_{opt}$ end

3 Lexical Grammar of micro C

1. **Lexical Elements**

   token:
   - *keyword*
   - *identifier*
   - *constant*
   - *string-literal*
   - *punctuator*

2. **Keywords**

   keyword: one of

return	**void**	**float**	**integer**
char	**for**	**const**	**while**
bool	**if**	**do**	**else**
begin	**end**		

3. **Identifiers**

   identifier:
   - *identifier-nondigit*
   - *identifier identifier-nondigit*
   - *identifier digit*

   identifier-nondigit: one of

_	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**
	n	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**	**x**	**y**	**z**
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
	N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**

   digit: one of

   0 1 2 3 4 5 6 7 8 9

4. **Constants**

constant:
    integer-constant
    floating-constant
    character-constant

integer-constant:
    nonzero-digit
    integer-constant digit
nonzero-digit: one of
    1 2 3 4 5 6 7 8 9
floating-constant:
    fractional-constant exponent-part$_{opt}$
    digit-sequence exponent-part
fractional-constant:
    digit-sequence$_{opt}$ **.** *digit-sequence*
    digit-sequence **.**
exponent-part:
    e *sign*$_{opt}$ *digit-sequence*
    E *sign*$_{opt}$ *digit-sequence*
sign: one of
    + −
digit-sequence:
    digit
    digit-sequence digit

' *c-char-sequence* **'**
c-char-sequence:
    c-char
    c-char-sequence c-char
c-char:
    any member of the source character set except
        the single-quote **'**, backslash ****, or new-line character
    escape-sequence
escape-sequence: one of
    **\' \" \? **
    \a \b \f \n \r \t \v

5. **String literals**

string-literal:
    " *s-char-sequence*$_{opt}$ **"**
s-char-sequence:
    s-char
    s-char-sequence s-char
s-char:
    any member of the source character set except
        the double-quote **"**, backslash ****, or new-line character
    escape-sequence

6. **Punctuators**

   punctuator: one of

   ```
   [ ]  ( )  ->  ++  --  &  *  +  -  !
   /  %  <<  >>  <  >  <=  >=  ==  !=  ^  |  &&  ||
   ?  :  ;  =  ,
   ```

7. **Comments**

   (a) *Multi-line Comment*

   Except within a character constant, a string literal, or a comment, the characters /* introduce a comment. The contents of such a comment are examined only to identify multibyte characters and to find the characters */ that terminate it. Thus, /* ... */ comments do not nest.

   (b) *Single-line Comment*

   Except within a character constant, a string literal, or a comment, the characters // introduce a comment that includes all multibyte characters up to, but not including, the next new-line character. The contents of such a comment are examined only to identify multibyte characters and to find the terminating new-line character.

4 Changes with respect to C

1. keywords are specified differently
2. Instead of braces { and }, we are using begin and end respectively.

5 The Assignment

1. Write a yacc specification for the language of micro C using the below grammar. The name of your file should be a7_roll.y for yacc file and a7_roll.l for lex file
2. Your code should also print a symbol table.
3. Your code should also print each grammar reduction.
4. Prepare a Makefile to compile the specifications and generate the lexer and parser.
5. Prepare a test input file a7_roll_test.mc that will test all the lexical rules and parser grammar rules that you have coded.
6. Prepare a zip file with the name a7_roll containing all the above files and upload to Moodle.

\ \ \ \\ \ \ \ \ \ \ \

Phrase Structure Grammar of microC for reference (examples)

1. **Expressions**:

/ The grammar is structured in a hierarchical way with precedences resolved. Associativity is handled by left or right recursion as appropriate.*/*

primary-expression:
    identifier *// Simple identifier*
    constant *// Integer or character constant*
    string-literal
    `(` *expression* `)`

postfix-expression: *// Expressions with postfix operators. Left assoc. in C; non-assoc. here*
    primary-expression
    postfix-expression `[` *expression* `]` *// 1-D array access*
    postfix-expression `(` *argument-expression-list*$_{opt}$ `)` *// Function invocation*
    postfix-expression `->` *identifier* *// Pointer indirection. Only one level*
    // Only a single postfix op is allowed in an expression here

argument-expression-list:
    assignment-expression
    argument-expression-list `,` *assignment-expression*

unary-expression:
    postfix-expression
    unary-operator unary-expression *// Expr. with prefix ops. Right assoc. in C; non-assoc. here*
    // Only a single prefix op is allowed in an expression here

unary-operator: one of
    `&` `*` `+` `-` `!` *// address op, de-reference op, sign ops, boolean negation op*

multiplicative-expression: *// Left associative operators*
unary-expression
multiplicative-expression `*` *unary-expression*
multiplicative-expression `/` *unary-expression*
multiplicative-expression `%` *unary-expression*

additive-expression: *// Left associative operators*
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

relational-expression: *// Left associative operators*
additive-expression
relational-expression `<` *additive-expression*
relational-expression `>` *additive-expression*
relational-expression `<=` *additive-expression*
relational-expression `>=` *additive-expression*

equality-expression: *// Left associative operators*
relational-expression
equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*

logical-AND-expression: *// Left associative operators*
equality-expression
logical-AND-expression `&&` *equality-expression*

logical-OR-expression: *// Left associative operators*
logical-AND-expression
logical-OR-expression `||` *logical-AND-expression*

conditional-expression: *// Right associative operator*
logical-OR-expression
logical-OR-expression `?` *expression* `:` *conditional-expression*

assignment-expression: *// Right associative operator*
conditional-expression
unary-expression `=` *assignment-expression* *// unary-expression must have lvalue*

expression:
assignment-expression

2. **Declarations** *declaration:* *// Simple identifier, 1-D array or function declaration of built-in type*
type-specifier init-declarator `;` *// Only one element in a declaration*

init-declarator:
declarator *// Simple identifier, 1-D array or function declaration*
declarator `=` *initializer* *// Simple id with init. initializer for array / fn/ is semantically skipped*

type-specifier: *// Built-in types*
`void`
`char`
`integer`

declarator:
$pointer_{opt}$ *direct-declarator* *// Optional injection of pointer*

direct-declarator:
identifier *// Simple identifier*
identifier `[` *integer-constant* `]` *// 1-D array of a built-in type or ptr to it. Only +ve constant*
identifier `(` $parameter\text{-}list_{opt}$ `)` *// Fn. header with params of built-in type or ptr to them*

pointer:
`*`

parameter-list:
parameter-declaration

parameter-list `,` *parameter-declaration*

parameter-declaration:

type-specifier $pointer_{opt}$ $identifier_{opt}$ *// Only simple ids of a built-in type or ptr to it as params*

initializer:

assignment-expression

3. **Statements**

statement:

compound-statement *// Multiple statements and / or nest block/s*

expression-statement *// Any expression or null statements*

selection-statement *// `if` or `if-else`*

iteration-statement *// `for`*

jump-statement *// `return`*

compound-statement:

`begin` $block\text{-}item\text{-}list_{opt}$ `end`

block-item-list:

block-item

block-item-list block-item

block-item: *// Block scope - declarations followed by statements*

declaration

statement

expression-statement:

$expression_{opt}$ `;`

selection-statement:

`if (` *expression* `)` *statement*

`if (` *expression* `)` *statement* `else` *statement*

iteration-statement:

`for (` $expression_{opt}$ `;` $expression_{opt}$ `;` $expression_{opt}$ `)` *statement*

jump-statement:

`return` $expression_{opt}$ `;`

4. **Translation Unit**

translation-unit: *// Single source file containing `main()`*

function-definition

declaration

function-definition:

type-specifier declarator `(` $declaration\text{-}list_{opt}$ `)` *compound-statement*

declaration-list:

declaration

declaration-list declaration